INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 820

INTRODUCING SYMBOLIC PROBLEM SOLVING TECHNIQUES IN THE DEPENDENCE TESTING PHASES OF A VECTORIZER

Alain LICHNEWSKY
François THOMASSET

AVRIL 1988

RR 820

Sur l' Introduction du Calcul Symbolique dans les Méthodes de Vectorisation Automatique

Introducing Symbolic Problem Solving Techniques in the Dependence Testing phases of a Vectorizer

Alain Lichnewsky
François Thomasset
I.N.R.I.A.
Domaine de Voluceau
78153 Le Chesnay CEDEX

Résumé

La tâche d'un *vectoriseur* est d'effectuer des transformations de programme susceptibles de mettre en évidence les boucles vectorielles les plus efficaces. Ce travail est guidé par une forme particulière d'analyse sémantique, le *Calcul des Dépendances*, qui doit être suffisamment précis pour exploiter complètement l'architecture vectorielle ou parallèle. En général les travaux sur la vectorisation réduisent cette phase à l'application d'une série de critères arithmétiques explicitement calculables. Il advient fréquemment que cette stratégie échoue, si les critères ne peuvent être calculés numériquement, et contiennent des symboles qui ne peuvent être évalués. Nous montrons ici qu'il est possible d'extraire des équations *symboliques* des critères classiques, d'en faire la synthèse avec d'éventuelles connaissances sur le programme, et d'utiliser le système obtenu pour décider de l'existence d'une dépendance. Dans le contexte du vectoriseur VATIL, il est également de contrôler le coût des calculs, obtenant ainsi globalement une efficacité satisfaisante.

Abstract

The purpose of a vectorizer is to perform program restructuring in order to exhibit the most efficiently exploitable forms of vector loops. This is guided by a suitable form of semantic analysis, *Dependence Testing*, which must be precise in order to fully exploit the architecture. Most studies reduce this phase to the application of a series of explicitely computable arithmetic criteria. In many cases, this will fail if the criteria cannot be computed numerically, and contain symbols which cannot be evaluated. This also makes the use of other information pertaining to these symbols difficult. It is shown here that it is possible to *extract symbolic equations* from some of the *classical criteria*, merge them with other symbolic knowledge about the program, and use the global system to decide the non-existence of a dependence. In the context of the VATIL vectorizer [17,16], it is also shown possible to control the computation cost, and obtain a good overall efficiency.

1 Introduction

In order to restructure loops for vector or parallel execution, it is essential to obtain very detailed information on the data and control dependences between multiply indexed occurrences of instructions. Roughly speaking, the original instructions are contained in a nested set of loops, which provide indexed sets of instruction occurrences. One looks for a different order of execution of these occurrences, permitting the parallel or vector execution of some loops, while retaining the original semantics.(Cf. [13],[17]). It is therefore essential that dependence analysis be capable to account for subscripted variables, and exploit available information on index variation and range.

Since the seminal work of D.J. Kuck and his team[13], many authors have approached this subject from several standpoints:

A Methods have been derived to determine subscripted data dependence, by reduction of subscript expressions, and loop index variations, to appropriate arithmetic criteria. (Cf. [2], [4], [3],[23]). The extension of such techniques to the cases where variable aliasing and reshaping are permitted is discussed in [6,22].

B Methods have been sought to collect globally the required semantic symbolic predicates in programs, from the standpoint of non-standard denotational semantics (Cf. [15]), or from an *ad-hoc* construction of sets of linearized predicates (Cf. [22]).

C General methods have been developped to solve symbolic equations occurring in the assessment of program semantics (Cf. Cousot [8]). To prove the theoretical results, a form of *monotonicity* is required. Further approximations are necessary since in the most general cases one is confronted with undecidable subproblems. We will discuss in the sequel how to exploit the information generated by such methods in the context of the vectorizer.

D Methods have been developped to solve decidable classes of sets of symbolic equations and inequations, as well as to *approximate* intractable [1] problems by simpler ones, which can be solved algorithmically, yielding some approximate information. (Cf. Bledsoe [5], Shostak[20]). The feasibility of approximation relies on the problem related fact that it is *safe* to replace a problem by a new one having a larger set of solutions, when the parallelization criteria possess the property of *monotonicity*.(Cf.[17]) The only price to be paid is that some vectorizable statements may not be vectorized.

Individually, each of these approaches have both strong advantages, and serious drawbacks, and we have found surprisingly few attempts to combine them in the litterature. Among these, the most notable are the following: Burke & Cytron[6] discuss the *class A* methods of Banerjee [4,3], Wolfe [23] and Allen [2], and the *class D* method of Shostak [20], but not their combination. Triolet[22] obtains semantic information localizing variables in *regions* [2], and maintains their polyhedral nature, but applies a simple *class D* solver [3] , to disprove directly the array-index aliasing equation (2). An apparently more general non-standart denotational

[1] The notion of *intractable problem* ranges from (potentially) undecidable ones, to problems whose solution is too costly.(Cf.[11,19,24])

[2] Unions of array element sets, each delimited by convex polyhedra in index space.

[3] Namely the Fourier-Motzkin Method [9]

semantic approach is taken by Jouvelot[15], but reduces to similar problem solving techniques.

The approach we have been using in the VATIL vectorizer attempts to combine the use of *class A* and *class D* methods. This is done by evaluating some of the classical criteria symbolically when it cannot be done explicitely numerically. At this point, we try to disprove the resulting set of predicates in the context of the available semantic information using *class D* approximate decision methods. Thus we are able to stay very close to well known criteria, which have been specifically optimized for the purpose of a vectorizer, and yet not limited by their classical numerically explicit implementation. Our current choice of *class D* problem solving procedure, is the **Sup-Inf** method of Bledsoe [5], which has a high level of generality, but already requires some form of "*computational complexity control*" in our environment [4] . Within the LeLisp [7] written implementation of VATIL, which makes heavy use of symbolic oriented routines, it has been easy to extend the existing programs to collect symbolic equations rather than signal failure to evaluate explicitely. Our techniques are extensible to make use of information collected by the methods of *classes B & C* (Cf.[8]), but we have no large scale experience on this [5] .

A major difficulty is related to the fact that the general symbolic systems which can be generated are potentially undecidable or impracticable because of their high computational cost. Thus, we rely on the fact that approximations can be made safely [17] , and on the adapted construction of the required symbolic systems. The overall efficiency of our approach is due to the use of costly general decision procedures only when classical criteria have failed, and even in such a case starting from well adapted generalized [6] versions of these specific arithmetic criteria. This is very different to the standpoint of the papers cited above, where it is proposed to start from scratch with the predicate decision procedure (Cf. [15,22]).

2 General Framework

2.1 Expressions and Predicates

We will be manipulating symbolic expressions and predicates in several contexts, and use the following notations:

- **variables** belong to the variable set $\mathcal{V}$ and represent elements of domain $\mathcal{D}$. We shall in the sequel use mnemonic variable names for application related variables like the *loop index* I.
- **constant names** represent elements of $\mathcal{D}$, the set of constants is noted $\mathcal{C}$.
- **expressions** are made up of variables and constants, combined with operators in $\mathcal{OP}$, they are elements of $\mathcal{E}$.
- **basic predicates** are formed from expressions using relation operators in $\mathcal{OR}$.
- **predicates** are formed from basic predicates using quantifier and logical operators, they form the set $\mathcal{PR}$.

We will often specify the context in which we will be working by the indication of $\{\mathcal{D},\mathcal{OP},\mathcal{OR}\}$. Moreover, we shall make use of

[4] Among our longer term plans, is the characterization of useful *class D* methods, adapted to the set of real problems.

[5] It is likely that the efficient selection of facts in the larger set of accumulated predicates will be a difficulty.

[6] The generalization involves embedding the (in) - equation where unevaluated variables and expressions remain, in a set of symbolic (in) - equations, after standardization and may be simplification.

abbreviations and some simple isomorphisms to avoid the rigidity of formal theory.

2.2 Program Fragments

In order to simplify the exposition we will restrain ourselves to a single loop, involving an explicit loop index I. This is not a constraint in the application of our methods. More precisely, let us consider the following loop:

```
C$PRAGMA{predicate-list}
        DO 1 I=1,N
:A:       X(exp1(I))= ....
:B:                 = X(exp2(I))      (1)
1       CONTINUE
```

where:

- exp1, exp2 are index expressions involving the loop index variable I, whereas N is an expression involving only loop invariant variables. These may have been found explicitely in the program, or may have been generated after loop standardisation, index linearization or aliasing explicitation (Cf.[6,1,17,13,6]). These expressions are made of constants, loop invariant integer variables , renamed variables assuming a single value [7] , and the loop index I. Since we will be creating new symbols as a result of several subalgorithms, we will denote by θ_k a set of unique symbols not initially in $\mathcal{V}$ ([8]).
- {predicate-list} is a list of **true** predicates obtained from some more global semantic analysis tool, involving the variables [9] in $\mathcal{V}$ and may be the loop index I. We are making use of two types of predicates: bounds :: $x \leq y$ and congruences [10] ::

$$x = y \bmod c \Longleftrightarrow \exists z \in \mathbf{Z}\ x = y + cz$$

- $\mathcal{F}$ is the set of current hypotheses, whose elements are predicates in $\mathcal{PR}$. Initially $\mathcal{F}$ is formed out of {predicate-list}, later on *known properties* and *necessary conditions* for the existence of a solution to the problem at hand will be added to it. Thus, if a contradiction is found, independence is proved.

A special function **AddP** is provided to add a predicate to the set of currently assumed predicates also noted $\mathcal{F}$. The function **NewS** creates new symbols in $\mathcal{V}$.

3 Dependence Analysis

A dependence exists between the statements :A: and :B: of (1), if: $\exists I \in \mathbf{Z}$, $\exists J \in \mathbf{Z}$:

$$exp1(I) = exp2(J) \quad (2)$$
$$I \diamond J \quad (3)$$
$$1 \leq I \ , \ J \leq N \quad (4)$$
$$\{predicate-list\} \models I, J \quad (5)$$

Where $\diamond$ can be $>$ (respectively $\leq$) in the case of an *anti-dependence* (resp. *data-dependence*) (Cf. [17,13]). The last equation simply means that I and J are values for I which satisfy the {predicate-list}. Of course, any other variable appearing in (2-5) is *implicitely bound* to an existential quantifier.

The exploitation of the precise knowledge of existing dependences is the major goal of

[7] Some precautions like this have to be taken when embedding our *program semantics problem* in a quite general mathematical setting

[8] and which are created by the meta-function " NewS ".

[9] Here also some precautions are to be taken when identifying program variables with symbols in $\mathcal{V}$

[10] here c is an integer constant

program restructuring. This permits vectorization ([17,13,12]), parallelization ([6,10], enhancement of data-locality ([14]),... Of course, the aim will most often be to prove that no dependence exists, and therefore that the system (2) has no solution. When solutions do exist, characterizing them in a more precise way can be of interest to apply more sophisticated techniques.

4 Dependence Criteria

After showing the basic steps on the example of the *direct resolution* of equation (2), we will recall two classical criteria, and detail the approach used to extract symbolic information from them when they fail to fully evaluate numerically. Since, we will show in the §5 that we can make use of such symbolic information, we will speak of *extended criteria*.

4.1 Index Equality Equation

The conceptually simplest method is to try to directly disprove equation (2). Although our implementation does not involve tackling directly this problem, but the *generalized criteria* shown below, we will describe now the direct approach to equation (2), many features of which will be of interest in the ensuing discussion.

We will use several devices to reduce the problem ($\mathcal{P}$) to a series of approximate ones ($\mathcal{P}'$), which satisfy the appropriate *safety condition*[11]:

$$Solutions(\mathcal{P}) \subseteq Solutions(\mathcal{P}') \qquad (6)$$

and then attempt to prove the last one has no solution. This can be done along the following steps:

[11] because of *monotonicity*, Cf.[17]

R1: Algebraically simplify equations (2,...,5), using for instance the techniques of Moses[18]. At the same time, the problem is reduced to integer arithmetic by observing that the expressions in (2) have integer values, thus non integer terms have to be converted. We add variables to achieve this:

$$\texttt{conv(expr)} \longrightarrow \theta := \texttt{NewS}$$

and forget about the remaining equation:

$$\theta = \texttt{conv(expr)}$$

It would be feasible to add predicates to take into account further properties:

$$\texttt{conv(sin(expr))} \longrightarrow -1 \leq \theta \leq 1$$

We also take the opportunity to get rid of any function or operator other than $+, *, min, max$, using the same mechanism. We are thus left with a problem in a $\{\mathbf{Z},(+,*),(\leq,=)\}$ first order theory, which is undecidable [19].

R2: select an approximate problem form which can be solved in practice. Among the candidates are:

- *C1:* Similar problems over the reals, in $\{\mathbf{R},(+,*),(\leq)\}$. Here, the problem has become decidable, but with exponential complexity. Moreover, even in very simple situations, working with the reals or rationals is not satisfactory for our application. This has led to the development of the *GCD*(§4.2) and *Exact Banerjee*[3] tests.
- *C2:* Linearized problems, and linear programming problems over the reals.
- *C3:* First order Presburger arithmetic in $\{\mathbf{Z},(+),(\leq,=)\}$ These problems can

theoretically be solved by exact decision methods which are extremely costly[12] or by approximate methods. *We have chosen such an approximate method, with restrictions on the allowed quantifiers, for our present implementation (Cf. 5.3).*

C4: Linearized problems, and linear programming problems over the integers. (Cf.[11])

R3: reduce the problem to the class selected. For instance, reduction to the class *C-9* means that variables in $\mathcal{V}$ will represent integers in **Z**, and involves replacing non-linear terms by a set of rules illustrated in figure 1.

Several facts must be noted about this approximation procedure. First of all, we *forget about* the old expressions whose values are represented by the newly introduced θ_i: we will not check that the initial problem indeed possesses a solution, but try to establish that the new approximate problem has no solution. Second, it is possible to enrich the predicate set $\mathcal{F}$ for the θ_i by using additional rules like available bounds for expressions. For instance, we could have added to the rule W1 (resp. W4) in the figure 1 the rule W5 (resp. W6 and W7) shown in the figure 2. At this point, we are left with an approximate problem $\mathcal{P}'$, to which the decision procedure of §5.3 is to be applied.

4.2 The GCD Test

This test assumes that the equation (2) is linear in the loop index:

$$(e_1, e_2, e) \subset \mathcal{E};\ e_1 I + e_2 J = e \qquad (7)$$

When $(e_1, e_2, e) \subset \mathcal{C}$, this test consists of observing that a necessary condition is:

$$e \equiv 0 \pmod{\gcd(e_1, e_2)} \qquad (8)$$

This is generalized by:

G1: *When it is determined* [13] *that* e_1 *and* e_2 *have a common constant factor a, either check the divisibility of e immediately or add the predicate:* $e \equiv 0 \pmod{a}$ *to* $\mathcal{F}$.

If required, the decision procedure described in paragraph 5.2 is then invoked.

4.3 The Allen - Banerjee - Wolfe Approximate Test

The linear form of equation (2) with respect to the loop index is also assumed here, and the context $\{\mathbf{Z},(+,*),(\leq,=)\}$ is used in the following derivation. A necessary condition for the existence of a solution to equations (2),(3),(4) is that the function $(I, J) \rightarrow e_1 I + e_2 J - e$ takes both signs when evaluated at the three corner of the triangle described by (3) and (4). This results in the lemma:

Lemma 1 *A necessary condition for equations (2),(3),(4) to have a solution is that the following holds: (Cf. [4,2,23])*

$$e_2 - (e_1^- - e_2)^+ (N-2) \leq e - e_1 - e_2 \qquad (9)$$

$$e - e_1 - e_2 \leq e_2 + (e_1^+ + e_2)^+ (N-2) \qquad (10)$$

The standard procedure is to check if one of the inequations can be completely evaluated and yields a contradiction. Our extension enable to consider the case where this does not occur and *symbolic* terms remain:

G2: *When (9) and (10) fail to evaluate and draw to a conclusion: first, simplify the expressions. Then choose the context*

[12] The minimal bound is of order $2^{2^{cn}}$ for the general case. More restricted classes have lower complexity in the (only) 2^{cn} range.

[13] Either by inspection of the expressions or using $\mathcal{F}$.

	Hypothesis	Rewrite rule	Side Effects
W1	$x, y \in \mathcal{E}$ $(x,y) \not\subset \mathbf{Z}$	$x * y \longrightarrow \theta$	θ:= NewS
W2	$x, y \in \mathcal{E}$ $x \div y \notin \mathbf{Z}$	$x \div y \longrightarrow \theta$	θ:= NewS
W3	$x \in \mathcal{E}$ $a \in \mathbf{N}$ $a \equiv 0 \mod 2$	$x^{**}a \longrightarrow \theta$	θ:= NewS AddP($1 \leq \theta$)
W4	$x \in \mathcal{E}$ $x \notin \mathbf{Z}$	$x^{+} \longrightarrow \theta$	θ := NewS AddP($0 \leq \theta$)

Figure 1: Linearizing for $\{Z,+,\leq\}$.

W5	$x, y \in \mathcal{E}$ $(x,y) \not\subset \mathbf{Z}$ {predicate-list}$\Rightarrow$ $0 \leq x*y \leq a$	$x * y \longrightarrow \theta$	θ:= NewS AddP($0 \leq \theta \leq a$)
W6	$x \in \mathcal{E}$ {predicate-list} $\Rightarrow x \geq 0$	$x^{+} \longrightarrow x$	
W7	$x \in \mathcal{E}$ {predicate-list} $\Rightarrow x \leq 0$	$x^{+} \longrightarrow 0$	

Figure 2: Improving Predicate List

$\{\mathbf{Z},(+),(\leq,=)\}$, linearize eventually the predicates in $\mathcal{F}$, and add equations (9) (10) after rewriting them according to the rules of figures 3 [14]*, 2 and 1.*

At this point we are left with a set of predicates to which the decision method of §5.3 can be applied.

[14]This ruleset is optional, see remark below

5 Decision Procedures for the Symbolic System

The main decision procedure we are using is the approximate method due to Bledsoe [5] and improved by Shostak[21]. As this last author pinpoints in [21, p. 534], this approximate method, and the reduction techniques for inequalities that it uses, make it well suited to test for problems that have solutions in the integers if and only if they have real solutions. This is very much in the same spirit as the approximate test of Allen-Banerjee-Wolfe (§4.3). However the divisibility test is inherently Dio-

W8	$x \in \mathcal{E}$	$x^+ \longrightarrow \max(x,0)$	$\max(x,0) \equiv \theta := \mathbf{NewS}$ $\mathrm{AddP}((\theta \geq 0)$ $\wedge((\theta = x) \vee (\theta = 0)))$

Figure 3: More precise treatment of Maxima

phantine, and we treat it differently.

5.1 Framework

First of all, we observe that we wish to prove predicate systems of quite particular forms, because of the origin of the problem, and the above listed transformations:

$$\forall v_1 \forall v_2 ... \forall v_i \quad \neg (D_1 \vee D_2 ... \vee D_l) \qquad (11)$$

$$\lambda = (1, ..., l) \quad D_\lambda = (B_1 \wedge ... \wedge B_{m_\lambda}) \qquad (12)$$

where the B_μ are basic predicates in $\mathcal{OR}$. Moreover, in most practical cases, the disjunction contains only one term D_1, and we have introduced so far only a single rule that may contribute to disjunctions in Figure 3. Of course, this is equivalent to proving the impossibility of:

$$\exists v_1 \exists v_2 ... \exists v_i \quad (D_1 \vee D_2 ... \vee D_l) \qquad (13)$$

5.2 Divisibility Criterion

In order to exploit the divisibility criterion **G1**, and take into account the information in $\mathcal{F}$, we simply use the basic predicates in $\mathcal{F}$ that can be put under the form:

$$v = expr \quad v \in \mathcal{V}$$

where the expression *expr* does not use variable v, to eliminate as many variables as possible in e of **G1**. At each step, the expression is simplified and tested for the divisibility criterion, until a contradiction is found.

5.3 The Sup-Inf Method

This method applies to predicate systems in $\{\mathbf{Z}, (+, \max, \min), (\leq)\}$, describing *quasi-linear* inequalities, of the form (11). Also, since we will be working in this framework, we could have avoided the rule W4, and the explicitation of max through quantifiers in rule W8 [15]. The special form involving only inequalities is achieved by effectively replacing equalities by the rules illustrated in Figure 4. It makes strong use of the fact that variables represent integers, and shows that the method is more oriented towards bounding the set of possible solutions of (13) in a (possibly empty) set, than to the proof of precise equalities among integers. Furthermore, to simplify the handling of constants, rational constants are allowed [16] in the computations and the estimates obtained by the method, but not in the reduction by W8 phase. Rational variables are not permitted.

We will not develop here the Sup-Inf Method with much detail, since this can be found in the original papers of Bledsoe [5]. The l subproblems of the disjunction are considered one at a time, and separately tested for solvability[17]. The test procedure $\mathcal{SI}$ consists of recursively calculating bounds for the variables.

Variables whose bounds are being calcu-

[15] This may be useful to achieve the requested form (11), also to avoid disjunctions. (Cf. §6.1

[16] This avoids having to constantly reduce to common denominator.

[17] The computational cost implication is clear, and we restrict the use of disjunction creating rules.

W9	$x \in \mathcal{E}$ $y \in \mathcal{E}$	$x = y \longrightarrow x \leq y \wedge y \leq x$
W10	$x \in \mathcal{E}$ $y \in \mathcal{E}$	$x < y \longrightarrow x + 1 \leq y$

Figure 4: From equalities to inequalities

```
      dimension a(1024)
c$PRAGMA ATTR (((<ge> k 2))) predicate)
c$PRAGMA ATTR (((<ge> j 2))) predicate)
      do 1 i =1,64
          a(1+(i-1)*64) = a (i+(i+k-1)*64)
     +        + a(j+(j-1)*64)
1     continue
```

```
      k1 = - 64+64*k
      k2 = - 64+(64*j+j)
C$DIRECTIVE SHORTLOOP,DOVEC
      DO 1 i = 1 ,64 ,1
         a(-63+64*i) = a(65*i+k1)+a(k2)
1     CONTINUE
```

Figure 5: Source and Vectorized Codes

lated are considered as *frozen* within the inner stages of the recursion. The procedure SI stops whenever an empty interval of the integers has been found, and otherwise returns precise bounds for the variables, in the reals, which are useless for our problem, since we do not want to go into the enumeration of possible integer values, because of computational cost. In the first favorable case, this implies that the approximate problem has no solution, and therefore that no dependence exists.

A practical alternative, proposed by Shostak[21] to the enumeration phase is to interpret the obtained bounds over the reals. According to the observation of [21, Theorem 17], the set of solutions S of (13) in the context [18] $\{\mathbf{R}, (+, \tilde{*}), (\leq)\}$ is convex and the bounds computed by the procedure SI are exact. It is possible to test for the emptiness of S by choosing iteratively arbitrary "slices" of S of decreasing dimensions and applying the above procedure SI in the reals. Whenever an empty interval[19] is found, it implies that S is empty, and we can conclude to the absence of dependence.

6 Implementation

6.1 Efficiency Issues

We basically recommand the use of our extended symbolic criteria after the failure of the well optimized classical tests, which avoids the issue of adding cost unnecessarily. In the VATIL current implementation[20], we avoid some bookkeeping by applying each extended test immediately upon failing to setup the corresponding classical test due to remaining symbolic terms. We first apply the *GCD* test,

[18] Here we use the symbol $\tilde{*}$ to denote multiplication by numeric constants in $\mathbf{R}$, since we cannot use the isomorphism with repeated addition here.

[19] over the Reals.

[20] This implementation is currently in use by the GIP-SM90 MVF Vectorizer.[16]

$\exists I\ \exists I1$	$-63+64*I$	$=$	$65*I1+K1$
	$I<I1$	;	$1\le I, I1\le 64$
Predicates:	$K\ge 2$	;	$J\ge 2$
Renaming :	$K1=-64+64*K$	;	$K2=-64+(64*J*J)$
$\exists K$	$-65*62\le$	$129+K1$	≤ 0
Sup Inf:	$K\ge 2$	;	$K\le -\frac{65}{64}$

Figure 6: Index Equality Equation and G2 Criterion

```
C***  KERNEL 2 ICCG EXCERPT (INCOMPLETE
      DO 200 L= 1,Loop
      IL= n
      IPNTP= 0
222   IPNT= IPNTP
      IPNTP= IPNTP+IL
      IL= IL/2
      i= IPNTP
c$PRAGMA ATTR (((i .  ipnt)) avail)
      DO 2 k= IPNT+2,IPNTP,2
        i= i+1
2       X(i)= X(k) - V(k)*X(k-1) - V(k+1)
     +         *X(k+1)
      IF( IL.GT.1) GO TO 222
200   CONTINUE
```

```
      DO 2 k11 = 1 ,1+(ipntp-(2+ipnt))/2
     +         ,256
       k2 = min0(256,
     +         (2+(ipntp-(2+ipnt))/2)-k11)
       k3 = -1+(ipnt+k11)
       k4 = -2+(2*k11+ipnt)
C$DIRECTIVE SHORTLOOP,DOVEC
      DO 10001 k1 = 1 ,k2 ,1
        x(k1+k3) = (x(2*k1+k4)-v(2*k1+k4)
     +  * x((-1+k4)+2*k1))-v((1+ k4)+2*k1)
     +  * x((1+k4)+2*k1)
10001CONTINUE
2     CONTINUE
      k = 2+(2*((ipntp-(2+ipnt))/2)+ipnt)
      i = 1+(ipnt+(ipntp-(2+ipnt))/2)
```

Figure 7: Source and Vectorized Codes

$\exists K1\ \exists L1$	$K1+k3$	$=$	$2*L1+k4$
	$K1<L1$	;	$1\le K1, L1\le k2$
Predicates:	$k2\le 256$	;	$k2\le (2+(ipntp-(2+ipnt))/2)-k11)$
	$1\le k11$	;	$k11\le 1+(ipntp-(2+ipnt))/2$
Renaming :			$k3=-1+k11+ipnt$
			$k4=-2+ipnt+2*k11$
$\exists k11$	$2+k11\le 0$	;	$0\le -2+2*k2+k11$
Sup Inf:	$\inf k2=2$	;	$\sup k2=256$
Sup Inf:$P_{\{k2=2\}}$	$\inf k11=1$	;	$\sup k11=-2$

Figure 8: Index Equality Equation and G2 Criterion

and then the *Allen - Banerjee - Wolfe* test. The extremely high potential cost of the symbolic decision procedure is alleviated by the fact that we do limit the amount of computer time given to the *Inf-Sup method*. Also, for the sake of efficiency we reduce the use of disjunction generating rules like W8 and W9. The first one is not necessary since max and min operators are allowed. For the second one, we observe that since we are working with linear equations, it is always possible to eliminate one variable from such an equation. This works globally well, and we plan to make a thorough experimental study of the efficiency matter for single and multiply nested loops before deciding of a more refined strategy.

Since we have implemented in VATIL the above strategy, we have been able to supress the use of the *exact Banerjee test* with no apparent loss of information[21] in processing meaningful programs. We also estimate that our present strategy is better suited for the study of both single and multiple loops.

The direct use of the *Index Equality Equation* is also an alternative which we have tried only on a very restricted set of examples, with poor performance compared to **G1** and **G2**.

6.2 Results

The figures 5, 7 and 9 show actual input and output from the current VATIL implementation. The figures 6 and 8 show the actual criterion used to decide the data dependence between the left hand side and the first term of the right hand side in these examples. These should enable the reader to figure out the type of predicate system which occurs in practice, and how the knowledge about program semantics is put to practical use by the VATIL software.

The application of the *generalized GCD* criterion **G1** is illustrated on figure (9). This furthermore illustrates the ability of our technique to take advantage of informations generated by algorithms studied by P. Cousot of *congruence type*.

The application of the *generalized Banerjee-Wolfe* criterion **G2** is illustrated on figures (5,6) and (7, 8). In the figure 8, one will notice that the method of testing emptiness of the solution set in the reals taking "slices" of lower dimension has been used, restricting to $k2 = 2$.

7 Conclusion

We have presented a method which permit the extension of classical dependence test in the direction of symbolic problem solving. This has been shown to be feasible while retaining the efficient classical tests and building upon existing sophisticated criteria. This approach has been shown capable to take into account information coming from semantic analyses for scalar variables, and to result in an efficient implementation. Naturally, the same process can be used in other similar areas of program analysis and restructuring.

Acknowledgments

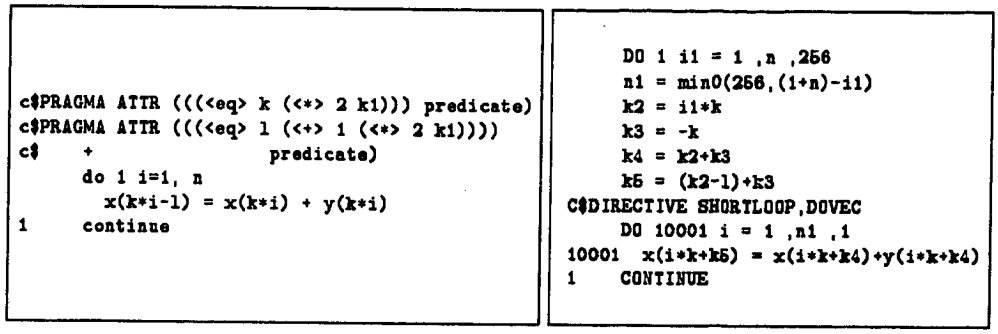
The authors are grateful to P. Flajolet and W. Jalby for helpful discussions, and B. Mélèse for careful proofreading.

References

[1] A. V. Aho, R. Sethi & J. D. Ullman, **"Compilers: Principles, Techniques and Tools"**, *Addison Wesley*, 1986.

[2] R. Allen, *"Dependence analysis for subscripted variables and its application*

[21] except on loops constructed *ad hoc*



```
c$PRAGMA ATTR ((((<eq> k (<*> 2 k1))) predicate)
c$PRAGMA ATTR ((((<eq> l (<+> 1 (<*> 2 k1))))
c$     +                  predicate)
       do 1 i=1, n
         x(k*i-l) = x(k*i) + y(k*i)
1      continue
```

```
       DO 1 i1 = 1 ,n ,256
       n1 = min0(256,(i+n)-i1)
       k2 = i1*k
       k3 = -k
       k4 = k2+k3
       k5 = (k2-1)+k3
C$DIRECTIVE SHORTLOOP,DOVEC
       DO 10001 i = 1 ,n1 ,1
10001  x(i*k+k5) = x(i*k+k4)+y(i*k+k4)
1      CONTINUE
```

Figure 9: Application of Extended GCD Test

to program transformations", Ph.D Dissertation, Dept. of Mathematical Sciences, Rice University, Houston, Texas, April 1983.

[3] U. Banerjee, *"Speed-up of ordinary programs", Phd. Thesis, Rep. 79-989, Dept. of Computer Science*, University of Illinois at Urbana-Champaign, October 1979.

[4] U.Banerjee, *"Data Dependence in Ordinary Programs", Rep 76-837, Dept. of Computer Science*, University of Illinois at Urbana-Champaign, November 1976.

[5] W.W.Bledsoe, *"A new Method for proving certain Presburger formulas", 4th Int. Joint Conf. Artif. Intell., Tbilissi*, September 1975, pp. 15-21.

[6] M. Burke, R. Cytron, *"Interprocedural Dependence Analysis and Parallelization", Proc. Compiler Construction Conf.*, 1986.

[7] J. Chailloux, *"LeLisp, Version 15.2: le manuel de référence", INRIA*, 1986.

[8] P.Cousot, *"Semantic Foundation of Program Analysis", in* S.S. Muchnick & N. D. Jones eds., **"Program Flow Analysis"**, Prentice Hall, 1981.

[9] P. Feautrier, A. Dumay, N. Tawbi, *"PAF: Un Parallèliseur Automatique pour Fortran", Rep. MASI, No 185, Université Paris 7*, 1987.

[10] D.D. Gajski, J.K. Peir, *"Toward computer aided programming for multiprocessors", in* M. Cosnard & al. ed., **"Parallel Algorithms & Architectures"**, North Holland, 1986.

[11] M. R. Garey, D. S. Johnson, **"Computers and Intractability"**, *Freeman and Co*, New York, 1979.

[12] K. Kennedy, *" Automatic translation of fortran programs to vector form", Rice U. Tech. Rep. 476-029-4*, 1980.

[13] D.J. Kuck, R.H. Kuhn, B. Leasure, M. Wolfe, *"The structure of an advanced retargetable vectorizer", in* Kai Hwang ed., **"Tutorial on Supercomputers: Design and Applications"**, IEEE Press , pp. 163-178, 1984.

[14] D. Gannon & W. Jalby, *"Strategies for Cache and Local Memory Management by Global Program Transforma-*

tion", Journal of Parallel and Distributed Computing, Special Issue, To appear.

[15] P. Jouvelot, *"Parallélisation Semantique ", Rep. MASI, No 174, Université Paris 7,* 1986.

[16] A. Lichnewsky, M. Loyer, *"Un Module Vectoriel Flottant sur SPS7. Pourquoi ?", Bulletin de Liaison de la Recherche en Informatique et en Automatique,* no 112, 1987.

[17] A. Lichnewsky, F. Thomasset, *"Techniques de base sur l'exploitation automatique du parallélisme dans les programmes", Rapport de Recherche INRIA,* n0. 460, Dec. 1985.

[18] J. Moses, *"Algebraic Simplification: A Guide for the Perplexed", Comm. ACM,* Vol 14 No 8, August 1971.

[19] M.O. Rabin, *"Decidable Theories", in* Barwise J. ed., **"Handbook of Mathematical Logic"**, North-Holland, 1977.

[20] R.E. Shostak, *"Deciding Linear Inequalities by Computing Loop Residues", JACM,* Vol 28, n0 4, October 1981, pp. 769-779.

[21] R.E. Shostak, *"On the Sup-Inf Method for Proving Presburger formulas", JACM,* Vol 24, n0 4, October 1977, pp. 529-543.

[22] R.J. Triolet, *"Interprocedural Analysis Based Program restructuring ", in* M. Cosnard & al. ed., **"Parallel Algorithms & Architectures"**, North Holland, 1986.

[23] M.J. Wolfe , *"Techniques for improving the inherent parallelism in programs", rep. UIUCDCS-R-78-929,* 1978.

[24] A. Yasuhara, **"Recursive Function Theory and Logic"**, *Academic Press,* 1977.

A Examples of VATIL output

The following set of examples shows the actual context of the standardized loops to which the previous dependence determination techniques are used, as well as some statistics.

Original Source Code

We show below the original source code, which will be submitted to a process of loop standardization and index variable extraction before the dependence can be computed. This seems not to be a very straightforward technique in simple examples where index expressions grow in complexity in this phase, but permits to handle aliasing and non explicit indexations. Since loop blocking is performed at this step, dependences that do not exist because the bounded index variation of the inner loop are taken into account by the general algorithm. In this particular case, the fact that the initial value of variable I is known to be IPNTP is used.

```
C*******************************************************************************
C***  KERNEL 2       ICCG EXCERPT (INCOMPLETE CHOLESKY - CONJUGATE GRADIENT)
C*******************************************************************************
C
      DO 200  L= 1,Loop
         IL= n
       IPNTP= 0
  222   IPNT= IPNTP
       IPNTP= IPNTP+IL
         IL= IL/2
          i= IPNTP
C$PRAGMA ATTR ( (i . ipntp) avail)
      DO 2 k= IPNT+2,IPNTP,2
          i= i+1
    2   X(i)= X(k) - V(k)*X(k-1) - V(k+1)*X(k+1)
        IF( IL.GT.1) GO TO 222
  200 CONTINUE
C
```

Standardized Loop

The loop shown below is submitted to the dependence analysis, as described in §4.2 and 4.3.

```
C*************************************************************************
C***  KERNEL 2       ICCG EXCERPT (INCOMPLETE CHOLESKY - CONJUGATE GRADIENT)
C*************************************************************************
C
      DO 200  L= 1,Loop
          IL= n
       IPNTP= 0
  222   IPNT= IPNTP
       IPNTP= IPNTP+IL
          IL= IL/2
           i= IPNTP
      DO 2 k11 = 1 ,1+(ipntp-(2+ipnt))/2 ,256
          k2 = min0(256,(2+(ipntp-(2+ipnt))/2)-k11)
          k3 = -1+(ipnt+k11)
          k4 = -2+(2*k11+ipnt)
C$DIRECTIVE SHORTLOOP
          DO 10001 ki = 1 ,k2 ,1
             x(k1+k3) = (x(2*k1+k4)-v(2*k1+k4)*x((-1+k4)+2*k1))-v((1+
     ! k4)+2*k1)*x((1+k4)+2*k1)
10001        CONTINUE
2         CONTINUE
          IF( IL.GT.1) GO TO 222
200     CONTINUE
C
```

Dependence Test Statistics

Statistics for the dependence tests using the extended criteria are shown below:

```
PREDICATE COMPUTATION STATISTICS:
DEPENDENCE TESTS :  3
EXTENDED   TESTS :  3
                                        TIME          SUCCESS
                                     MAX      MEAN
DIRECT INDEX EQUALITY EQN:          21.7      19.9        0  %
GENERALIZED ALLEN BANERJEE:         7.42      5.38      100  %
COMBINATION INDEX EQ, GEN A.B.:     29.9      26.9       67  %
```

In this case the strategy of using the Generalized Allen Banerjee Wolfe test of §4.3 is more efficient than the Direct Test of §4.1. Success rate shows the percentage of cases when the algorithm has been able to prove independence. The smaller rate of success of the combined test is due to the fact that we limit the total available time for the symbolic testing.

Standardized Loop

```
C
C*************************************************************************
C***  KERNEL 8      A.D.I. INTEGRATION
C*************************************************************************
C
      DO  8      L = 1,Loop
               nl1 = 1
               nl2 = 2
      DO  8     kx = 2,3
C$DIRECTIVE PRAGMA nl1 = 1, nl2 = 2
C      DO 8 ky11 = 1 ,-1+n ,256
         k1 = min0(256,n-ky11)
         k2 = 1+kx
         k3 = 1+ky11
C$DIRECTIVE SHORTLOOP
         DO 10007 ky1 = 1 ,k1 ,1
            du1(ky1+ky11) = u1(kx,k3+ky1,1)-u1(kx,(-2+k3)+ky1,1)
            du2(ky1+ky11) = u2(kx,k3+ky1,1)-u2(kx,(-2+k3)+ky1,1)
            du3(ky1+ky11) = u3(kx,k3+ky1,1)-u3(kx,(-2+k3)+ky1,1)

            u1(kx,ky1+ky11,2) = (((a11*du1(ky1+ky11)+u1(kx,ky1+ky11,1))
     ! +a12*du2(ky1+ky11))+a13*du3(ky1+ky11))+sig*(u1(-2+k2,ky1+ky11,1)+
     ! (u1(k2,ky1+ky11,1)-2.000000e+00*u1(kx,ky1+ky11,1)))

            u2(kx,ky1+ky11,2) = (((a21*du1(ky1+ky11)+u2(kx,ky1+ky11,1))
     ! +a22*du2(ky1+ky11))+a23*du3(ky1+ky11))+sig*(u2(-2+k2,ky1+ky11,1)+
     ! (u2(k2,ky1+ky11,1)-2.000000e+00*u2(kx,ky1+ky11,1)))

            u3(kx,ky1+ky11,2) = (((a31*du1(ky1+ky11)+u3(kx,ky1+ky11,1))
     ! +a32*du2(ky1+ky11))+a33*du3(ky1+ky11))+sig*(u3(-2+k2,ky1+ky11,1)+
     ! (u3(k2,ky1+ky11,1)-2.000000e+00*u3(kx,ky1+ky11,1)))
10007       CONTINUE
8     CONTINUE
```

Dependence Test Statistics

```
PREDICATE COMPUTATION STATISTICS:
DEPENDENCE TESTS :  39
EXTENDED   TESTS :  18
                                        TIME            SUCCESS
                                    MAX      MEAN
DIRECT INDEX EQUALITY EQN:         5.04      1.92         66  %
GENERALIZED ALLEN BANERJEE:        4.10      0.76         66  %
COMBINATION INDEX EQ, GEN A.B.:    5.08      1.28         66  %
```

B Dependence Computation Statistics

The following statistics were obtained on a larger sample of 24 loops, and give an indication of the typical behaviour of the extended criteria. The success rate is not shown, since it can only be interpreted on a dependence per dependence basis: failure to prove independence when a true dependence exists should not be blamed.

```
PREDICATE COMPUTATION STATISTICS:
DEPENDENCE TESTS :401
EXTENDED   TESTS : 46
                                        TIME
                                    MAX      MEAN
DIRECT INDEX EQUALITY EQN:         21.8      3.00
GENERALIZED ALLEN BANERJEE:        7.42      0.80
COMBINATION INDEX EQ, GEN A.B.:    29.9      3.04
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique